



Technical Document 2179
December 1990

Proposed Standard for a Generic Package of Primitive Functions for Ada

Draft 1.0

ISO-IEC/JTC1/SC22/WG9 (Ada)
Numerics Rapporteur Group

G. Myers, Chair



Approved for public release; distribution is unlimited.

19950227 173

Technical Document 2179

December 1990

Proposed Standard for a Generic Package of Primitive Functions for Ada

Draft 1.0

ISO-IEC/JTC1/SC22/WG9 (Ada)
Numerics Rapporteur Group

G. Myers, Chair

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Date	
Approved by Codes	
Dist	Avail and/or Special
A-1	

Generic Package of Primitive Functions (GPPF) for Ada

Description

The proposed standard for the Generic Package of Primitive Functions (GPPF) for Ada represents the work of a large number of people in both the United States and Europe who have collaborated to develop specifications for packages of Ada mathematical functions. This development has been difficult and lengthy. The exceptional dedication and perseverance of these people have resulted in the completed specifications for two packages—GPPF, and the Generic Package of Elementary Functions (GPEF) for Ada.

GPPF is the specification for primitive functions and procedures for manipulating the fraction part and exponent part of machine numbers of the generic floating-point type. Additional functions are provided for directed rounding to a nearby integer, for computing an exact remainder, for determining the immediate neighbors of a floating-point machine number, for transferring the sign from one floating-point machine number to another, and for shortening a floating-point machine number to a specified number of leading radix digits.

Background

The Ada-Europe Numerics Working Group (A-ENWG) was formed in 1984 about the same time that an early study proposing standard mathematical packages in Ada was undertaken by Symm and Kok. In 1986, the Numerics Working Group (NUMWG) of the Association of Computing Machinery's Special Interest Group on Ada (ACM SIGAda) was formed, and has met every few months since. During the 1980s, members of A-ENWG met on a regular basis with the NUMWG so that close cooperation was achieved on developing specifications that were joint effort of both groups. The A-ENWG has not met for some three years, but the NUMWG continues informal liaison with key European Ada individuals on continuing work.

Current Status of Standardization

The proposed standards for GPEF and GPPF have been adopted by the Numerics Rapporteur Group (NRG), a subcommittee of Working Group 9 (Ada) of Subcommittee 22 of Joint Technical Committee 1 of the International Organization for Standardization-International Electrotechnical Commission (ISO-IEC JTC1/SC22/WG9 (Ada)). WG9 has approved both proposed standards and has forwarded them to SC22 for voting. GPEF has been accepted as Draft International Standard (DIS) 11430 and GPPF has been approved as DIS 11729. The completion of editorial formatting of both documents for final publication as international standards is expected this year.

Gilbert Myers
Chair, ACM SIGAda NUMWG, ISO NRG
May 10, 1994

CONTENTS

1.	PURPOSE	1
2.	SUBPROGRAMS PROVIDED	1
3.	INSTANTIATIONS	1
4.	IMPLEMENTATIONS	2
5.	MACHINE NUMBERS AND STORABLE MACHINE NUMBERS	3
6.	DENORMALIZED NUMBERS	4
7.	EXCEPTIONS	4
8.	SPECIFICATIONS OF THE SUBPROGRAMS	5
8.1	EXPONENT—EXPONENT OF THE CANONICAL REPRESENTATION OF A FLOATING-POINT MACHINE NUMBER	5
8.2	FRACTION—SIGNED MANTISSA OF THE CANONICAL REPRESENTATION OF A FLOATING-POINT MACHINE NUMBER	5
8.3	DECOMPOSE—EXTRACT THE COMPONENTS OF THE CANONICAL REPRESENTATION OF A FLOATING-POINT MACHINE NUMBER	6
8.4	COMPOSE—CONSTRUCT A FLOATING-POINT MACHINE NUMBER FROM THE COMPONENTS OF ITS CANONICAL REPRESENTATION	6
8.5	SCALE—INCREMENT/DECREMENT THE EXPONENT OF THE CANONICAL REPRESENTATION OF A FLOATING-POINT MACHINE NUMBER	7
8.6	FLOOR—GREATEST INTEGER NOT GREATER THAN A FLOATING- POINT MACHINE NUMBER, AS A FLOATING-POINT NUMBER	7
8.7	CEILING—LEAST INTEGER NOT LESS THAN A FLOATING-POINT MACHINE NUMBER, AS A FLOATING-POINT NUMBER	8
8.8	ROUND—INTEGER NEAREST TO A FLOATING-POINT MACHINE NUMBER, AS A FLOATING-POINT NUMBER	8
8.9	TRUNCATE—INTEGER PART OF A FLOATING-POINT MACHINE NUMBER, AS A FLOATING-POINT NUMBER	9
8.10	REMAINDER—EXACT REMAINDER UPON DIVIDING ONE FLOATING-POINT MACHINE NUMBER BY ANOTHER	9
8.11	ADJACENT—FLOATING-POINT MACHINE NUMBER NEXT TO ONE FLOATING-POINT MACHINE NUMBER IN THE DIRECTION OF A SECOND	10
8.12	SUCCESSOR—FLOATING-POINT MACHINE NUMBER NEXT ABOVE A GIVEN FLOATING-POINT MACHINE NUMBER	10
8.13	PREDECESSOR—FLOATING-POINT MACHINE NUMBER NEXT BELOW A GIVEN FLOATING-POINT MACHINE NUMBER	11
8.14	COPY_SIGN—TRANSFER OF SIGN FROM ONE FLOATING-POINT MACHINE NUMBER TO ANOTHER	11

8.15	LEADING_PART—FLOATING-POINT MACHINE NUMBER WITH ITS MANTISSA (IN THE CANONICAL REPRESENTATION) TRUNCATED TO A GIVEN NUMBER OF RADIX-DIGITS	12
	RATIONALE FOR THE PROPOSED STANDARD FOR A GENERIC PACKAGE OF PRIMITIVE FUNCTIONS FOR Ada	13
	REFERENCES	20

This document defines the specification of a generic package of primitive functions and procedures called **GENERIC_PRIMITIVE_FUNCTIONS**. It does not define the body.

Working closely with the Ada-Europe Numerics Working Group, the ACM SIGAda Numerics Working Group has prepared this document for submission to the ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group as a proposed standard. It is intended to provide primitive operations required to endow mathematical software, such as implementations of the elementary functions, with the qualities of accuracy, efficiency, and portability. With this standard, such mathematical software can achieve all of these qualities simultaneously; without it, one or more of them typically must be sacrificed.

The generic package specification included in this document is presented as compilable Ada and is followed by explanatory text in numbered sections. The explanatory text is an integral part of the standard, with the exception of the following items:

- (1) in Section 8, notes (under the heading *Notes* associated with some of the subprograms); and
- (2) examples and notes (labeled as such) presented at the end of any numbered section.

As used in this document, "must" and "shall" both express a requirement; "may" expresses permission; "should" expresses a recommendation; and "can," "might," and "could" all express possibility. In formulas, $[v]$ and $\lceil v \rceil$ mean the greatest integer less than or equal to v , and the least integer greater than or equal to v , respectively, and other notations have their customary meaning.

This proposal was prepared under the leadership of G. Myers with contributions by the following individuals, listed in alphabetical order: A. Adamson, J.-M. Chebat, W. J. Cody, P. M. Cohen, S. G. Cohen, T. J. Dekker, R. B. K. Dewar, K. W. Dritz, B. Ford, J. B. Goodenough, G. S. Hodgson, J. Kok, R. F. Mathis, T. G. Mattson, B. T. Smith, J. S. Squire, P. T. P. Tang, W. A. Whitaker, and D. T. Winter. Many others contributed through international meetings and electronic mail reviews. Organizations lending support to this effort were the Naval Ocean Systems Center, Argonne National Laboratory, Westinghouse Electric Corporation, Numerical Algorithms Group, Centrum voor Wiskunde en Informatica, Software Productivity Consortium, Contel, Martin Marietta, the Software Engineering Institute (Carnegie Mellon University), Quantitative Technology Corporation, Alslys, the Courant Institute of Mathematical Sciences (New York University), University of Amsterdam, and IBM.

Bibliography

- [1] W. S. Brown. A Simple but Realistic Model of Floating-Point Computation. *TOMS*, 7(4):445-480, December 1981.
- [2] W. S. Brown and S. I. Feldman. Environment Parameters and Basic Functions for Floating-Point Computation. *TOMS*, 6(4):510-523, December 1980.
- [3] K. W. Dritz. Rationale for the Proposed Standard for a Generic Package of Primitive Functions for Ada. ANL Report ANL-90/41, Argonne National Laboratory, Argonne, Illinois, December 1990.
- [4] B. Ford. Parameterization of the Environment for Transportable Mathematical Software. *TOMS*, 4(2):100-103, June 1978.
- [5] B. Ford, J. Kok, and M. W. Rogers, editors. *Scientific Ada*. Cambridge University Press, Cambridge, 1986.
- [6] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.
- [7] IEEE. IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std. 854-1987, IEEE, New York, 1987.
- [8] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, December 1990. Draft 1.2.
- [9] J. Kok. Proposal for Standard Mathematical Packages in Ada. CWI Report NM-R8718, Centrum voor Wiskunde en Informatica, Amsterdam, November 1987.
- [10] R. F. Mathis. Elementary Functions Packages for Ada. In *Proc. 1987 ACM SIGAda International Conference on the Ada Programming Language* (special issue of *Ada Letters*), pages 95-100, December 1987.

- [11] U. S. Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A. U. S. Government Printing Office, Washington, D. C., 1983. Also adopted by ISO as ISO/8652-1987, Programming Languages—Ada.

```

generic
  type FLOAT_TYPE is digits <>;
  type EXPONENT_TYPE is range <>;
package GENERIC_PRIMITIVE_FUNCTIONS is

  function EXPONENT (X : FLOAT_TYPE) return EXPONENT_TYPE;
  function FRACTION (X : FLOAT_TYPE) return FLOAT_TYPE ;
  procedure DECOMPOSE (X : in FLOAT_TYPE;
                       FRACTION : out FLOAT_TYPE;
                       EXPONENT : out EXPONENT_TYPE) ;
  function COMPOSE (FRACTION : FLOAT_TYPE;
                   EXPONENT : EXPONENT_TYPE) return FLOAT_TYPE ;
  function SCALE (X : FLOAT_TYPE;
                 EXPONENT : EXPONENT_TYPE) return FLOAT_TYPE ;

  function FLOOR (X : FLOAT_TYPE) return FLOAT_TYPE ;
  function CEILING (X : FLOAT_TYPE) return FLOAT_TYPE ;
  function ROUND (X : FLOAT_TYPE) return FLOAT_TYPE ;
  function TRUNCATE (X : FLOAT_TYPE) return FLOAT_TYPE ;
  function REMAINDER (X, Y : FLOAT_TYPE) return FLOAT_TYPE ;

  function ADJACENT (X, TOWARDS : FLOAT_TYPE) return FLOAT_TYPE ;
  function SUCCESSOR (X : FLOAT_TYPE) return FLOAT_TYPE ;
  function PREDECESSOR (X : FLOAT_TYPE) return FLOAT_TYPE ;

  function COPY_SIGN (VALUE, SIGN : FLOAT_TYPE) return FLOAT_TYPE ;
  function LEADING_PART (X : FLOAT_TYPE;
                       RADIX_DIGITS : POSITIVE) return FLOAT_TYPE ;

end GENERIC_PRIMITIVE_FUNCTIONS;

```

1. Purpose

This generic package contains primitive functions and procedures for manipulating the fraction part and the exponent part of machine numbers (cf. Section 5) of the generic floating-point type. Additional functions are provided for directed rounding to a nearby integer, for computing an exact remainder, for determining the immediate neighbors of a floating-point machine number, for transferring the sign from one floating-point machine number to another, and for shortening a floating-point machine number to a specified number of leading radix-digits. Some subprograms are redundant in that they are combinations of other subprograms. This is intentional so that convenient calls and fast execution can be provided to the user.

These subprograms are intended to augment standard Ada operations and to be useful in portably implementing such packages as those providing real and complex elementary functions, where (for example) the steps of argument reduction and result construction demand fast, error-free scaling and remaindering operations.

2. Subprograms provided

The following fifteen subprograms are provided:

EXPONENT	FRACTION	DECOMPOSE	COMPOSE	SCALE
FLOOR	CEILING	ROUND	TRUNCATE	REMAINDER
ADJACENT	SUCCESSOR	PREDECESSOR		
COPY_SIGN	LEADING_PART			

The EXPONENT and FRACTION functions, and the DECOMPOSE procedure, decompose a floating-point machine number into its constituent parts, whereas the COMPOSE function constructs a floating-point machine number from those parts. The SCALE function scales a floating-point machine number accurately by a power of the hardware radix. The FLOOR, CEILING, ROUND, and TRUNCATE functions all yield an integer value (in floating-point format) "near" the given floating-point argument, using distinct methods of rounding. The REMAINDER function provides an accurate remainder for floating-point operands, using the semantics of the IEEE REM operation. The ADJACENT, SUCCESSOR, and PREDECESSOR functions yield floating-point machine numbers in the immediate vicinity of other floating-point machine numbers. The COPY_SIGN function transfers the sign of one floating-point machine number to another. The LEADING_PART function retains only the specified number of high-order radix-digits of a floating-point number, effectively replacing the remaining (low-order) radix-digits by zeros.

3. Instantiations

This standard describes a generic package which the user must instantiate to obtain a package. The generic package has two generic formal parameters: a generic formal type named FLOAT_TYPE and a generic formal type named EXPONENT_TYPE. At instantiation, the user must specify a floating-point type or subtype as the generic actual parameter to be associated with FLOAT_TYPE, and an integer type or subtype as the generic actual parameter to be associated with EXPONENT_TYPE. These are referred to below as the "generic actual types." These types are used as the parameter and, where applicable, the result types of the subprograms contained in the generic package.

Depending on the implementation, the user may or may not be allowed to associate, with FLOAT_TYPE, a generic actual type having a range constraint (cf. Section 4). If allowed, such a range constraint will have the usual effect of causing CONSTRAINT_ERROR to be raised when a floating-point argument outside the user's range is passed in a call to one of the subprograms, or when one of the subprograms attempts to return a floating-point value (either as a function result or as a formal parameter of mode out) outside the user's range. Allowing the generic actual type associated with FLOAT_TYPE to have a range constraint also has some implications for implementors.

The user is allowed to associate any integer-type generic actual type with EXPONENT_TYPE. However, insufficient range in the generic actual type will have the usual effect of causing CONSTRAINT_ERROR to be raised

when an integer-type argument outside the user's range is passed in a call to one of the subprograms, or when one of the subprograms attempts to return an integer-type value (either as a function result or as a formal parameter of mode out) outside the user's range. Further considerations are discussed in Section 4.

In addition to the body of the generic package itself, implementors may provide (non-generic) library packages that can be used just like instantiations of the generic package for the predefined floating-point types (in combination with `INTEGER` for `EXPONENT_TYPE`). The name of a package serving as a replacement for an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS` in which `FLOAT_TYPE` is equated with `FLOAT` (and `EXPONENT_TYPE` with `INTEGER`) should be `PRIMITIVE_FUNCTIONS`; for `LONG_FLOAT` and `SHORT_FLOAT`, the names should be `LONG_PRIMITIVE_FUNCTIONS` and `SHORT_PRIMITIVE_FUNCTIONS`, respectively; etc. When such a package is used in an application in lieu of an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS`, it must have the semantics implied by this standard for an instantiation of the generic package. This standard does not prescribe names for implementor-supplied non-generic library packages serving as pre-instantiations of `GENERIC_PRIMITIVE_FUNCTIONS` for `EXPONENT_TYPES` other than `INTEGER`.

4. Implementations

For the most part, the results specified for the subprograms in Section 8 do not permit the kinds of approximations allowed by Ada's model of floating-point arithmetic. For this reason, portable implementations of the body of `GENERIC_PRIMITIVE_FUNCTIONS` are not believed to be possible. An implementation of the standard in Ada may use pragma `INTERFACE` or other pragmas, unchecked conversion, machine-code insertions, representation clauses, or other machine-dependent techniques as desired.

An implementor is assumed to have knowledge of the underlying hardware environment and is expected to utilize that knowledge to produce the exact results (or, in a few cases, highly constrained approximations) specified by this standard; for example, implementations may directly manipulate the exponent field and fraction field of floating-point numbers.

An implementation is allowed to impose a restriction that the generic actual type associated with `FLOAT_TYPE` must not have a range constraint that reduces the range of allowable values. If it does impose this restriction, then the restriction must be documented, and the effects of violating the restriction must be one of the following:

- (1) Compilation of a unit containing an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS` is rejected.
- (2) `CONSTRAINT_ERROR` or `PROGRAM_ERROR` is raised during the elaboration of an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS`.

Conversely, if an implementation does not impose the restriction, then it must not allow such a range constraint, when included with the user's actual type, to interfere with the internal computations of the subprograms; that is, if the floating-point argument and result are within the range of the type, then the implementation must return the result and must not raise an exception (such as `CONSTRAINT_ERROR`).

An implementation must not allow insufficient range in the user's generic actual type associated with `EXPONENT_TYPE` to interfere with the internal computations of a subprogram when the range is sufficient to accommodate the integer-type arguments and integer-type results of the subprogram.

An implementation must function properly in a tasking environment. Apart from the obvious restriction that an implementation of `GENERIC_PRIMITIVE_FUNCTIONS` must avoid declaring variables that are global to the subprograms, no special constraints are imposed on implementations. Nothing in this standard requires the use of such global variables.

Some hardware and their accompanying Ada implementations have the capability of representing and discriminating between positively and negatively signed zeros as a means, e.g., of preserving the sign of an infinitesimal quantity that has underflowed to zero. This standard allows implementations of `GENERIC_PRIMITIVE_FUNCTIONS` to exploit that capability, when available, in appropriate ways. At the same time, it accommodates implementations in which that capability is unavailable. Because a definition of what comprises the capability of representing and distinguishing signed zeros is beyond the scope of this standard,

implementations are allowed the freedom not to exploit the capability, even when it is available. This standard accommodates the various choices allowed to implementations by, e.g., defining the results of `COPY_SIGN` to be an implementation dependent choice between two values when its `SIGN` argument is zero. An implementation must exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in this package. In addition, an implementation must document its behavior with respect to signed zeros.

Note:

It is intended that implementations of `FLOOR`, `CEILING`, `ROUND`, and `TRUNCATE` determine the result without an intermediate conversion to an integer type, which might raise an exception.

5. Machine numbers and storable machine numbers

In the broad sense, a floating-point “machine number” of type `FLOAT_TYPE` is any number that can arise in the course of computing with operands and operations of that type. The set of such numbers depends on the implementation of Ada. Some implementations hold intermediate results in extended registers having a longer fraction part and/or wider exponent range than the storage cells that hold the values of variables. Thus, in the broad sense, there can be two or more different representations of floating-point machine numbers of type `FLOAT_TYPE`.

One such representation is that of the set of “storable” floating-point machine numbers. This representation is assumed to be the one characterized by the representation attributes of `FLOAT_TYPE`—for example (and in particular), `FLOAT_TYPE'MACHINE_MANTISSA`, `FLOAT_TYPE'BASE'FIRST`, and `FLOAT_TYPE'BASE'LAST`. The significance of the storable floating-point machine numbers is that they can be assumed to be propagated by assignment, parameter association, and function returns; because of the limited lifetime of values held in extended registers, there is no guarantee that a floating-point machine number outside this subset, once generated, can be so propagated.

The machine numbers referred to subsequently in this document are to be understood to be storable floating-point machine numbers. An implementation of `GENERIC_PRIMITIVE_FUNCTIONS` is thus entitled to assume that the arguments of all of its subprograms are always storable floating-point machine numbers; furthermore, to support this standard, an implementation of Ada must guarantee that only storable floating-point machine numbers are received as arguments by these subprograms. Without the assumption and the restriction, the exact results specified by this standard would be unrealistic (because, for example, they would imply that extra-precise results must be delivered when extra-precise arguments are received), and those specified for `ADJACENT`, `SUCCESSOR`, and `PREDECESSOR` would not even be well-defined.

The storability of a subprogram's arguments does not always guarantee that the desired mathematical result is representable as a storable floating-point machine number. In the few subprograms where the desired mathematical result can sometimes be unrepresentable, the actual result is permitted to be a specified approximation of the mathematical result, or it is omitted and replaced by the raising of an exception (cf. Section 7).

The term “neighboring machine number” is used in two contexts in this standard.

- (1) When a desired mathematical result α is not representable but lies within the range of machine numbers, it necessarily falls between two adjacent machine numbers, the one immediately above and the one immediately below; those two numbers are referred to as the “machine numbers neighboring α .”
- (2) Every machine number X except the most positive (`FLOAT_TYPE'BASE'LAST`) has a nearest neighbor in the positive direction, and every one except the most negative (`FLOAT_TYPE'BASE'FIRST`) has a nearest neighbor in the negative direction; each is referred to as the “machine number neighboring X ” in the given direction.

In both cases, the identity of the neighboring machine numbers is uniquely (if here only informally) determined by the fact that the set of machine numbers is understood to be the set of storable machine numbers (having `FLOAT_TYPE'MACHINE_MANTISSA` radix-digits in the fractional part of their canonical form) and is totally ordered.

6. Denormalized numbers

On machines fully or partially obeying IEEE arithmetic, the denormalized numbers are included in the set of machine numbers if the implementation of Ada uses the hardware in such a way that they can arise from normal Ada arithmetic operations (such implementations are said in this standard to “recognize denormalized numbers”); otherwise, they are not. Whether an implementation recognizes denormalized numbers determines whether the results of some subprograms, for particular arguments, are exact or approximate; it is also taken into account in determining the results that can be produced by the `ADJACENT`, `SUCCESSOR`, and `PREDECESSOR` functions.

As used in this standard, a nonzero quantity α is said to be “in the denormalized range” when $|\alpha| < \text{FLOAT_TYPE}'\text{MACHINE_RADIX}^{*(\text{FLOAT_TYPE}'\text{MACHINE_EMIN}-1)}$; the term “canonical form of a floating-point number” is taken from the Ada Reference Manual, but its applicability is here extended to denormalized numbers by allowing the leading digit of the fractional part to be zero when the exponent part is equal to `FLOAT_TYPE'``MACHINE_EMIN`.

7. Exceptions

Various conditions can make it impossible for a subprogram in `GENERIC_PRIMITIVE_FUNCTIONS` to deliver a result. Whenever this occurs, the subprogram raises an exception instead. No exceptions are declared in `GENERIC_PRIMITIVE_FUNCTIONS`; thus, only predefined exceptions are raised, as described below.

The `REMAINDER` function performs an operation related to division. When its second argument is zero, it raises the exception specified by Ada for signaling division by zero (this is `NUMERIC_ERROR` in the Ada Reference Manual, but it is changed to `CONSTRAINT_ERROR` by AI-00387).

The result defined for the `SCALE`, `COMPOSE`, `SUCCESSOR`, `PREDECESSOR`, and, on some hardware, `COPY_SIGN` functions can exceed the overflow threshold of the hardware. When this occurs (or, more precisely, when the defined result exceeds `FLOAT_TYPE'``BASE'``LAST` in magnitude), the function raises the exception specified by Ada for signaling overflow (this is `NUMERIC_ERROR` in the Ada Reference Manual, but it is changed to `CONSTRAINT_ERROR` by AI-00387).

All of the subprograms, as stated in Section 3, are subject to raising `CONSTRAINT_ERROR` when an integer-type value outside the bounds of the user’s generic actual type associated with `EXPONENT_TYPE` is passed as an argument, or when one of the subprograms attempts to return such an integer-type value. Similarly, if the implementation allows range constraints in the generic actual type associated with `FLOAT_TYPE`, then `CONSTRAINT_ERROR` will be raised when the value of a floating-point argument lies outside the range of that generic actual type, or when a subprogram in `GENERIC_PRIMITIVE_FUNCTIONS` attempts to return a value outside that range. Additionally, all of the subprograms are subject to raising `STORAGE_ERROR` when they cannot obtain the storage they require.

Whereas a result that is too large to be represented causes the signaling of overflow, a result that is too small to be represented exactly does *not* raise an exception; such a result, which can be computed by `SCALE`, `COMPOSE`, and `REMAINDER`, is instead approximated (possibly by zero), as specified separately for each of these subprograms.

The only exceptions allowed during an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS`, including the execution of the optional sequence of statements in the body of the instance, are `CONSTRAINT_ERROR`, `PROGRAM_ERROR`, and `STORAGE_ERROR`, and then only for the reasons given below. The raising of `CONSTRAINT_ERROR` during instantiation is only allowed when the implementation imposes the restriction that the generic actual type associated with `FLOAT_TYPE` must not have a range constraint, and the user violates that restriction (it may, in fact, be an inescapable consequence of the violation). The raising of `PROGRAM_ERROR` during instantiation is only allowed for the purpose of signaling errors made by the user—for example, violation of this same restriction. The raising of `STORAGE_ERROR` during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

8. Specifications of the subprograms

Except where an approximation is explicitly allowed and defined, the formulas given below under the heading *Definition* specify precise mathematical results. In a few cases, these formulas leave a subprogram undefined for certain arguments; in those cases, the subprogram will raise an exception, as stated under the heading *Exceptions*, instead of delivering a result.

In the specifications of EXPONENT, FRACTION, DECOMPOSE, COMPOSE, SCALE, and LEADING_PART, the symbol β stands for the value of FLOAT_TYPE'MACHINE_RADIX.

8.1. EXPONENT—Exponent of the Canonical Representation of a Floating-Point Machine Number

Specification:

```
function EXPONENT (X : FLOAT_TYPE) return EXPONENT_TYPE;
```

Definition:

- (a) $\text{EXPONENT}(0.0) = 0$
- (b) For $X \neq 0.0$, $\text{EXPONENT}(X) =$ the unique integer k such that $\beta^{k-1} \leq |X| < \beta^k$

Notes:

When X is a denormalized number, $\text{EXPONENT}(X) < \text{FLOAT_TYPE'MACHINE_EMIN}$.

8.2. FRACTION—Signed Mantissa of the Canonical Representation of a Floating-Point Machine Number

Specification:

```
function FRACTION (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

- (a) $\text{FRACTION}(0.0) = 0.0$
- (b) For $X \neq 0.0$, $\text{FRACTION}(X) = X \cdot \beta^{-k}$, where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$

8.3. DECOMPOSE—Extract the Components of the Canonical Representation of a Floating-Point Machine Number

Specification:

```
procedure DECOMPOSE (X          : in  FLOAT_TYPE;
                     FRACTION : out FLOAT_TYPE;
                     EXPONENT  : out EXPONENT_TYPE);
```

Definition:

- (a) FRACTION = 0.0 and EXPONENT = 0 upon return from an invocation of DECOMPOSE(0.0, FRACTION, EXPONENT)
- (b) For $X \neq 0.0$, FRACTION = $X \cdot \beta^{-k}$ and EXPONENT = k , where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$, upon return from an invocation of DECOMPOSE(X, FRACTION, EXPONENT)

Notes:

When X is a denormalized number, EXPONENT < FLOAT_TYPE'MACHINE_EMIN upon return from an invocation of DECOMPOSE(X, FRACTION, EXPONENT).

8.4. COMPOSE—Construct a Floating-Point Machine Number from the Components of its Canonical Representation

Specification:

```
function COMPOSE (FRACTION : FLOAT_TYPE;
                  EXPONENT : EXPONENT_TYPE) return FLOAT_TYPE;
```

Definition:

- (a) COMPOSE(0.0, EXPONENT) = 0.0 for any EXPONENT
- (b) For FRACTION $\neq 0.0$, let $\alpha = \text{FRACTION} \cdot \beta^{\text{EXPONENT}-k}$, where k is the unique integer such that $\beta^{k-1} \leq |\text{FRACTION}| < \beta^k$. If α is exactly representable as a floating-point machine number (cf. Section 5), COMPOSE(FRACTION, EXPONENT) = α ; otherwise, COMPOSE(FRACTION, EXPONENT) = either one of the machine numbers neighboring α , provided that $|\alpha| < \text{FLOAT_TYPE'BASE'LAST}$.

Exceptions:

When α as defined above is such that $|\alpha| > \text{FLOAT_TYPE'BASE'LAST}$, COMPOSE raises the exception specified by Ada for signaling overflow (cf. Section 7) instead of delivering a result.

Notes:

- (a) For $\text{FRACTION} \neq 0.0$, this function can deliver an approximation (possibly zero) to the exact mathematical result α only when EXPONENT is sufficiently negative to force α to be in the denormalized range, and either the implementation does not recognize denormalized numbers, or α is not exactly representable as a denormalized number (cf. Section 6).
- (b) The name **FRACTION** is not meant to suggest that the first argument is restricted to fractional values; rather, it is meant to suggest that the first argument supplies (via its fractional part in the canonical form) the fractional part of the result.

8.5. SCALE—Increment/Decrement the Exponent of the Canonical Representation of a Floating-Point Machine Number

Specification:

```
function SCALE (X          : FLOAT_TYPE;  
                EXPONENT : EXPONENT_TYPE) return FLOAT_TYPE;
```

Definition:

Let $\alpha = X \cdot \beta^{\text{EXPONENT}}$. If α is exactly representable as a floating-point machine number (cf. Section 5), $\text{SCALE}(X, \text{EXPONENT}) = \alpha$; otherwise, $\text{SCALE}(X, \text{EXPONENT}) =$ either one of the machine numbers neighboring α , provided that $|\alpha| < \text{FLOAT_TYPE}'\text{BASE}'\text{LAST}$.

Exceptions:

When α as defined above is such that $|\alpha| > \text{FLOAT_TYPE}'\text{BASE}'\text{LAST}$, **SCALE** raises the exception specified by Ada for signaling overflow (cf. Section 7), instead of delivering a result.

Notes:

This function can deliver an approximation (possibly zero) to the exact mathematical result α only when EXPONENT is sufficiently negative to force α to be in the denormalized range, and either the implementation does not recognize denormalized numbers, or α is not exactly representable as a denormalized number (cf. Section 6).

8.6. FLOOR—Greatest Integer Not Greater Than a Floating-Point Machine Number, as a Floating-Point Number

Specification:

```
function FLOOR (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

$$\text{FLOOR}(X) = \lfloor X \rfloor$$

Notes:

For sufficiently large $|X|$, this function merely returns its argument.

8.7. CEILING—Least Integer Not Less Than a Floating-Point Machine Number, as a Floating-Point Number

Specification:

```
function CEILING (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

$$\text{CEILING}(X) = \lceil X \rceil$$

Notes:

For sufficiently large $|X|$, this function merely returns its argument.

8.8. ROUND—Integer Nearest to a Floating-Point Machine Number, as a Floating-Point Number

Specification:

```
function ROUND (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

$\text{ROUND}(X) =$ the integer nearest to X ; if X is equidistant from two integers, then the even integer is chosen

Notes:

For sufficiently large $|X|$, this function merely returns its argument.

8.9. TRUNCATE—Integer Part of a Floating-Point Machine Number, as a Floating-Point Number

Specification:

```
function TRUNCATE (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

$$\text{TRUNCATE}(X) = \begin{cases} \lfloor X \rfloor, & X \geq 0.0 \\ \lceil X \rceil, & X < 0.0 \end{cases}$$

Notes:

For sufficiently large $|X|$, this function merely returns its argument.

8.10. REMAINDER—Exact Remainder Upon Dividing One Floating-Point Machine Number by Another

Specification:

```
function REMAINDER (X, Y : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

For $Y \neq 0.0$, let $\alpha = X - (Y \cdot n)$, where n is the integer nearest to the exact value of X/Y ; if $|n - (X/Y)| = 1/2$, then n is chosen to be even. If α is exactly representable as a floating-point machine number (cf. Section 5), $\text{REMAINDER}(X, Y) = \alpha$; otherwise, $\text{REMAINDER}(X, Y) = 0.0$.

Exceptions:

For any X , $\text{REMAINDER}(X, 0.0)$ raises the exception specified by Ada for signaling division by zero (cf. Section 7) instead of delivering a result.

Notes:

- (a) This function can deliver an approximation (namely, zero) to the exact mathematical result α only when Y is in the neighborhood of zero, X is sufficiently close to a multiple of Y to force α to be in the denormalized range, and the implementation does not recognize denormalized numbers (cf. Section 6).
- (b) The magnitude of the result is $\leq |Y/2|$.

8.11. ADJACENT—Floating-Point Machine Number Next to One Floating-Point Machine Number in the Direction of a Second

Specification:

```
function ADJACENT (X, TOWARDS : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

- (a) $\text{ADJACENT}(X, X) = X$
- (b) For $\text{TOWARDS} \neq X$, $\text{ADJACENT}(X, \text{TOWARDS})$ = the floating-point machine number (cf. Section 5) neighboring X in the direction toward TOWARDS ; in an implementation exploiting signed zeros (cf. Section 4), a zero result must have the sign of X

Notes:

- (a) Unlike `SUCCESSOR` and `PREDECESSOR`, to which it is related, `ADJACENT` never raises an exception.
- (b) For certain normalized arguments, the numerical value of the result depends on whether or not the implementation recognizes denormalized numbers (cf. Section 6). For example, for $\text{TOWARDS} \neq 0.0$, $\text{ADJACENT}(0.0, \text{TOWARDS})$ yields a denormalized number if the implementation recognizes denormalized numbers, and a normalized number otherwise. Similarly, $\text{ADJACENT}(\pm \sigma, 0.0)$, where σ is the smallest positive normalized number, yields a denormalized number if the implementation recognizes denormalized numbers, and zero otherwise.

8.12. SUCCESSOR—Floating-Point Machine Number Next Above a Given Floating-Point Machine Number

Specification:

```
function SUCCESSOR (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

$\text{SUCCESSOR}(X)$ = the floating-point machine number (cf. Section 5) neighboring X in the positive direction, provided that $X \neq \text{FLOAT_TYPE}'\text{BASE}'\text{LAST}$; in an implementation exploiting signed zeros (cf. Section 4), a zero result must be a negatively signed zero

Exceptions:

Since there is no floating-point machine number neighboring $\text{FLOAT_TYPE}'\text{BASE}'\text{LAST}$ in the positive direction (a consequence of the assumption and restriction in Section 5), `SUCCESSOR` raises the exception specified by Ada for signaling overflow (cf. Section 7), instead of delivering a result, when $X = \text{FLOAT_TYPE}'\text{BASE}'\text{LAST}$.

Notes:

For certain arguments, the numerical value of the result depends on whether or not the implementation recognizes denormalized numbers (cf. Section 6). For example, `SUCCESSOR(0.0)` yields a denormalized number if the implementation recognizes denormalized numbers, and a normalized number otherwise. Similarly, `SUCCESSOR(- σ)`, where σ is the smallest positive normalized number, yields a denormalized number if the implementation recognizes denormalized numbers, and zero otherwise.

8.13. PREDECESSOR—Floating-Point Machine Number Next Below a Given Floating-Point Machine Number

Specification:

```
function PREDECESSOR (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

`PREDECESSOR(X)` = the floating-point machine number (cf. Section 5) neighboring `X` in the negative direction, provided that `X` \neq `FLOAT_TYPE'BASE'FIRST`; in an implementation exploiting signed zeros (cf. Section 4), a zero result must be a positively signed zero

Exceptions:

Since there is no floating-point machine number neighboring `FLOAT_TYPE'BASE'FIRST` in the negative direction (a consequence of the assumption and restriction in Section 5), `PREDECESSOR` raises the exception specified by Ada for signaling overflow (cf. Section 7), instead of delivering a result, when `X = FLOAT_TYPE'BASE'FIRST`.

Notes:

For certain arguments, the numerical value of the result depends on whether or not the implementation recognizes denormalized numbers (cf. Section 6). For example, `PREDECESSOR(0.0)` yields a denormalized number if the implementation recognizes denormalized numbers, and a normalized number otherwise. Similarly, `PREDECESSOR(σ)`, where σ is the smallest positive normalized number, yields a denormalized number if the implementation recognizes denormalized numbers, and zero otherwise.

8.14. COPY_SIGN—Transfer of Sign from One Floating-Point Machine Number to Another

Specification:

```
function COPY_SIGN (VALUE, SIGN : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

- (a) For $VALUE \neq 0.0$, $COPY_SIGN(VALUE, SIGN) = \begin{cases} |VALUE|, & SIGN > 0.0 \\ \pm|VALUE| \text{ (see note a)}, & SIGN = 0.0 \\ -|VALUE|, & SIGN < 0.0 \end{cases}$
- (b) $COPY_SIGN(0.0, SIGN) = 0.0$ (see note b)

Exceptions:

Since the negation of some representable values causes overflow on some hardware (e.g., when 2's-complement representation is used for floating-point), $COPY_SIGN$ raises the exception specified by Ada for signaling overflow (cf. Section 7), instead of delivering a value, in that case.

Notes:

- (a) Two cases arise when $VALUE \neq 0.0$ and $SIGN = 0.0$:
- (i) in an implementation exploiting signed zeros (cf. Section 4), $COPY_SIGN$ must deliver $-|VALUE|$ when $SIGN$ is a negatively signed zero and $|VALUE|$ when $SIGN$ is a positively signed zero;
 - (ii) in an implementation not exploiting signed zeros, $COPY_SIGN$ must deliver $|VALUE|$.
- (b) In an implementation exploiting signed zeros, the zero delivered by $COPY_SIGN(0.0, SIGN)$ must be a negatively signed zero when either $SIGN < 0.0$ or $SIGN$ is a negatively signed zero, and it must be a positively signed zero when either $SIGN > 0.0$ or $SIGN$ is a positively signed zero.

8.15. LEADING_PART—Floating-Point Machine Number with its Mantissa (in the Canonical Representation) Truncated to a Given Number of Radix-Digits

Specification:

```
function LEADING_PART (X           : FLOAT_TYPE;
                       RADIX_DIGITS : POSITIVE) return FLOAT_TYPE;
```

Definition:

- (a) $LEADING_PART(0.0, RADIX_DIGITS) = 0.0$ for any $RADIX_DIGITS$
- (b) For $X > 0.0$, $LEADING_PART(X, RADIX_DIGITS) = \lfloor X/\beta^{k-RADIX_DIGITS} \rfloor \cdot \beta^{k-RADIX_DIGITS}$, where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$
- (c) For $X < 0.0$, $LEADING_PART(X, RADIX_DIGITS) = \lceil X/\beta^{k-RADIX_DIGITS} \rceil \cdot \beta^{k-RADIX_DIGITS}$, where k is the unique integer such that $\beta^{k-1} \leq |X| < \beta^k$

Notes:

For $RADIX_DIGITS \geq FLOAT_TYPE'MACHINE_MANTISSA$, this function merely returns its first argument.

Rationale for the Proposed Standard for a Generic Package of Primitive Functions for Ada

Kenneth W. Dritz

December 1990

Abstract

This paper supplements the "Proposed Standard for a Generic Package of Primitive Functions for Ada," written by the ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Based on recommendations made jointly by the ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group, the proposed primitive functions standard is the second of several anticipated secondary standards to address the interrelated issues of portability, efficiency, and robustness of numerical software written in Ada. Its purpose, features, and developmental history are outlined in this commentary.

At about the time that work on a proposed Ada standard for the elementary functions began in 1986, early efforts to implement the elementary functions—square root, logarithm, trigonometric functions, and the like—underscored the need to be able to perform certain steps in their computation with extreme accuracy. These functions are typically implemented by transforming the argument so that it lies within a reduced range, computing the desired function on the transformed argument by a polynomial or rational approximation (designed to be sufficiently accurate over the relatively narrow reduced argument range) to obtain an intermediate result, and then constructing the final result by appropriately transforming the intermediate result. Accuracy is controlled in the middle step by the choice of approximation method, which bounds the approximation error. However, the final result can be extremely sensitive to errors (such as roundoff errors) made in the argument reduction step. Unnecessary error can also enter in the final step if the transformation it represents is not carried out carefully.

Details of the transformations needed in the argument reduction and result construction steps depend, of course, on the function being implemented. In the case of the periodic functions, the essential requirement is to compute an accurate remainder when the argument is divided by the period, if specified; when the period is allowed to default to the irrational 2π , a technique other than a simple division is required to obtain a suitably accurate remainder. In other cases, especially SQRT and LOG, decomposition of the argument into its exponent and fraction parts is the starting point, with the fraction part (or a simple function of it) becoming the transformed argument; the result construction step in these cases usually involves a simple modification—often just a scaling—of the intermediate result by a simple function of the exponent part.

If one is interested in implementing the elementary functions in a portable fashion, how does one go about computing accurate floating-point remainders and decomposing floating-point numbers into their constituent parts portably? Two problems arise if one tries to do these things entirely in portable Ada: the result is inefficient, often involving loops that require many traversals; and it cannot be proven fully accurate with Ada's model of floating-point arithmetic, since the model caters to the weaknesses of the weakest conforming implementation of Ada. (On machines manifesting them, such weaknesses—for example, lack of a guard digit—can introduce errors in the argument reduction step that become amplified as the loops are traversed.) The efficiency and accuracy problems can be solved, of course, by judicious use of representation clauses or interface programming in assembler language or even machine language insertions, given knowledge of the host machine, but that obviously destroys portability.

Exact floating-point remainder and decomposition of a floating-point number into its constituent parts are two examples of *primitive functions*—low-level floating-point functions having the property that they cannot be coded in Ada so as to be simultaneously accurate, efficient, and portable. Since we know how to solve the accuracy and efficiency problems when details of the underlying machine are available (indeed, some of the primitive functions are directly available as hardware operations on specific machines), all that is really lacking is a standardized interface to the functions. That is what the proposed generic primitive functions standard [1] provides.

Portable implementations of the generic elementary functions standard will be the first beneficiary of the generic primitive functions standard; others will follow. However, the generic primitive functions standard will always have a specialized clientele: experts, probably highly trained numerical analysts, concerned with the development of high-quality, portable mathematical software. It is not for the average application programmer.

The proposed standard has been developed by the ACM SIGAda Numerics Working Group in collaboration with the Ada-Europe Numerics Working Group. The proposal has been adopted by the WG9 Numerics Rapporteur Group and is to be submitted to WG9 and its parents, leading ultimately to an ISO standard. The standardization effort has been supported and encouraged in the United States by the Ada Joint Program Office of the U.S. Department of Defense, and in Europe by the Commission of the European Communities.

Although work on the primitive and the elementary functions standards began at about the same time, the elementary functions standard was completed, except for some late refinements, about a year and a half earlier [9]. Earliest drafts of the primitive functions standard drew heavily from recommendations made many years earlier in [3]; other works influencing the Ada primitive functions at an early date were [6, 11, 15, 14]. Later versions of the primitive functions were influenced by the IEEE floating-point standards [7, 8] and by the proposed Language Compatible Arithmetic Standard (LCAS) [12, 13]. One reason for the delay in completing the primitive functions, relative to the elementary functions, was a series of late additions to the proposed primitive functions standard as the result of evolving implementation experience with the elementary functions. Another was the recognition that software intending to exploit IEEE arithmetic had to pay particular attention to some of its more subtle features, such as denormalized numbers and signed zeros. It took considerable effort to describe the primitive functions so that they could be implemented in either IEEE or non-IEEE environments. This issue also had ramifications for the elementary functions standard, resulting in a recent revision of it [10] and in the updating of its rationale document [5].

The proposed standard for the primitive functions defines the specification of a generic package called `GENERIC_PRIMITIVE_FUNCTIONS`. It is a package because that is the accepted way to collect together several related subprograms; it is generic, with generic formal parameters for the two types used for the arguments and results of the subprograms, in view of the rules for parameter associations and the inability to anticipate the types used in applications. The generic formal parameter `FLOAT_TYPE` gives the type to be used for the floating-point arguments and results of subprograms in `GENERIC_PRIMITIVE_FUNCTIONS`, while the generic formal parameter `EXPONENT_TYPE` gives the type to be used for the few integer arguments and results that, with one exception,¹ deal with exponents of the canonical machine representation. When an instantiation of `GENERIC_PRIMITIVE_FUNCTIONS` is used in an implementation of the elementary functions (e.g., in the body of `GENERIC_ELEMENTARY_FUNCTIONS`), the `FLOAT_TYPE` of the latter should be passed through to the former, and a sufficiently wide integer type should be associated with `EXPONENT_TYPE`. The predefined type `INTEGER` probably suffices for the latter, but if one is worried about sufficient range, then an integer type whose range covers `SYSTEM.MIN_INT .. SYSTEM.MAX_INT` can be defined and used instead.

Like the elementary functions standard, the primitive functions standard permits implementations to impose a restriction that the generic actual type associated with `FLOAT_TYPE` in an instantiation must not contain a range constraint that reduces the range of allowable values. Implementations choosing not to impose the restriction must be designed to be immune from the avoidable effects of such range constraints; in general, this means that variables of type `FLOAT_TYPE` cannot safely be used for intermediate results within an implementation of `GENERIC_PRIMITIVE_FUNCTIONS`. Those imposing the restriction must document it; they can safely use such variables, but they must behave in one of several stated ways (i.e., predictably) if the restriction is violated. (For a detailed discussion of the genesis of this optional restriction and its implications, see the latest revision of [5]. The freedom of an implementation to impose the restriction will be revoked in the future if and when Ada—for example, as part of the Ada 9X revision process—acquires additional functionality that allows the declaration of variables having the precision, but not the range, of a generic formal floating-point type.) Incidentally, implementations are not allowed to impose a similar restriction on the generic actual types that can be associated with `EXPONENT_TYPE` during instantiation; it is not difficult to implement `GENERIC_PRIMITIVE_FUNCTIONS` to be efficient while limiting the consequences of insufficient range in that generic actual type to the unavoidable raising of `CONSTRAINT_ERROR` during a subprogram call or return.

¹One of the subprograms takes an argument that is a nonzero count of the number of digits to be retained in a particular computation; the predefined integer subtype `POSITIVE` is used for the corresponding parameter.

Perhaps the most significant difference between the two standards, other than subject area, is their respective handling of accuracy requirements. The elementary functions standard allowed implementations to approximate the exact mathematical result but constrained the approximation error by requiring implementations to satisfy "maximum relative error" bounds. In contrast, the primitive functions standard requires implementations to deliver the exact mathematical result defined for each function, whenever that result is representable; approximations are permitted only when the mathematical result is not representable and is smaller in magnitude than the smallest normalized positive floating-point number; and even then, the result is constrained to be one of the adjacent representable numbers. This level of accuracy is an essential aspect of the definition of the functions as operations on machine numbers yielding related machine numbers, without which their utility in argument reduction, etc., would be compromised. Achieving the required accuracy is *not* something that can be accomplished portably in Ada, at least not without making assumptions about the performance of the hardware that go well beyond the requirements imposed by the Ada model of floating-point arithmetic. On the other hand, the required accuracy can be easily and efficiently achieved by targeting implementations for specific environments and by utilizing knowledge of the machine representations in conjunction with appropriate operations (often integer or bit operations), accessed if necessary through low-level interfaces. A precedent for the accuracy required of the primitive functions can be found in the Ada attribute T'BASE'LAST for a floating-point type T: by definition, it has full machine-number accuracy, which, in general, exceeds model-number and safe-number accuracy.

Because the primitive functions transform machine numbers into other well-defined machine numbers, the standard includes a discussion of exactly what is meant by "floating-point machine number" within the context of the subprograms' definitions. What numbers are in the set of machine numbers? Does that set include the extra-precise numbers that some Ada implementations generate as a consequence of using extended registers for intermediate results? The answer to the latter question must be no, for otherwise the precise mathematical formulas used to specify the results of some of the functions would imply that the output from a function must be extra-precise if its input is, and yet the programmer has no means to ensure that that will be the case. Thus, it is clearly stated that the "machine numbers" referred to throughout the standard are the *storable* machine numbers—the ones that can be (a) stored; (b) propagated by assignment, parameter association, and function returns; and (c) characterized by the representation attributes FLOAT_TYPE'MACHINE_MANTISSA, FLOAT_TYPE'BASE'FIRST, and FLOAT_TYPE'BASE'LAST. Implementations of the primitive functions are entitled to assume that only storable machine numbers will be seen as arguments, and implementations of Ada must uphold that assumption (by forcing storage, if necessary, before calling a primitive function) in order for implementations of the primitive functions to have any hope of conforming to the standard.

Furthermore, because some hardware (e.g., that implementing IEEE arithmetic) has the capability of representing denormalized numbers—those with the minimum exponent and an unnormalized fraction part—one must also be precise about whether the set of machine numbers includes them. The standard says that it does if the hardware has the capability of representing them and the Ada implementation uses the hardware in such a way that it actually generates them; otherwise, it does not. This is especially significant when talking about "adjacent machine numbers," since the machine number adjacent to the smallest positive normalized number, in the direction toward zero, will be a denormalized number if the hardware and Ada implementation recognize denormalized numbers, and zero otherwise. It is also germane to the approximations that are permitted when a defined result falls in the denormalized range and is not exactly representable.

Some hardware (again, typically hardware conforming to IEEE arithmetic) has the capability of representing both positive and negative zeros (i.e., the sign of zero is relevant in some contexts). Like the elementary functions standard, the primitive functions standard allows signed zeros to be exploited if they are present in the hardware, but does not require them to be exploited. And like the elementary functions standard, the primitive functions standard does not give the required sign of each zero result (when signed zeros are being exploited), but leaves that to other standards or interpretations.² The behavior of one of the primitive functions, COPY_SIGN, does depend on the sign of a zero argument (when signed zeros are being exploited), as is also true of ARCTAN and ARCCOT in the elementary functions. The standard also clarifies that plus and minus zero are *not* to be considered "adjacent" (and therefore different) machine numbers, in any context where adjacency is relevant; thus, the "neighbors" of zero do not depend on the sign of zero.

²There are four exceptions, however. The required signs of zero results from ADJACENT, SUCCESSOR, PREDECESSOR, and COPY_SIGN are spelled out in the standard because those functions are intimately concerned with representations.

Early versions of the proposed primitive functions standard did not permit *any* approximations: when the exact mathematical result was not representable, they called for the raising of an exception to signal that fact. Indeed, this applied not just to underflow situations,³ but to overflow as well. An exception called `REPRESENTATION_ERROR` was reserved for that purpose. Commenting on an early version of the proposal, an observer convinced the committee that it would be better to signal overflow in the usual way (i.e., by raising the predefined exception provided by Ada for that contingency) and that it would also be better to provide a result conforming to the Ada standard in cases of underflow (including flushing to zero, if nothing better could be done) instead of raising an exception. An overflow or underflow in the result of a primitive function is most likely to occur when the primitive function is used for scaling purposes in the final step of some other computation, such as that of an elementary function. In such a case, the elementary function would overflow or underflow as well, and it would be undesirable to force the latter to intercept a `REPRESENTATION_ERROR` exception arising in the former just so that it could substitute some other behavior. As the primitive functions standard is now written, an overflow or underflow occurring in the result of a primitive function called to perform scaling in the final step of the computation of an elementary function can simply be propagated from the primitive function through the elementary function to the latter's caller, which will thus satisfy the requirements of the elementary functions standard in a most efficient way.

With underflows reported by approximations and overflows signaled by the appropriate predefined exception, there was no longer any need for the `REPRESENTATION_ERROR` exception, which was accordingly eliminated. No exceptions are declared by `GENERIC_PRIMITIVE_FUNCTIONS`. Only predefined exceptions may be raised by implementations of the primitive functions, and even those are restricted (as they were in the elementary functions standard) to specific cases where they are unavoidable.

The subprograms (fourteen functions and one procedure) in `GENERIC_PRIMITIVE_FUNCTIONS` can be organized into four groups for presentation purposes. In the discussions that follow, arguments and results are of the floating-point type `FLOAT_TYPE` except where noted, and β stands for the value of `FLOAT_TYPE'MACHINE_RADIX`.

The first group comprises basic decomposition, composition, and scaling subprograms for floating-point numbers. These are the `EXPONENT`, `FRACTION`, `COMPOSE`, and `SCALE` functions and the `DECOMPOSE` procedure.

`EXPONENT` is primarily useful in argument reduction steps, where it gives a coarse indication of the magnitude of its argument. Except when $X = 0.0$, the function `EXPONENT(X)` delivers—as a value of the integer type `EXPONENT_TYPE`—the unique integer k such that $\beta^{k-1} \leq |X| < \beta^k$. This definition is entirely mathematical and not related to the representation of X on the machine. Thus, as a positive X decreases through the normalized range and into the denormalized range, `EXPONENT(X)` continues to decrease, even though the exponent part of the machine representation of X stops decreasing when the smallest normalized number is reached. In fact, on the assumption that `FLOAT_TYPE'MACHINE_EMIN` is the value of that minimum exponent,⁴ `EXPONENT(X)` can return a value that is less than `FLOAT_TYPE'MACHINE_EMIN` (e.g., when X is denormalized). Finally, `EXPONENT(0.0)` is defined in this standard to deliver 0.

The `EXPONENT` function can be computed on typical hardware by extracting and unbiasing the exponent field of the representation, with a special case for $X = 0.0$ and with additional steps required when X is denormalized. `EXPONENT` corresponds closely to the IEEE recommended function `logb`, which is usually available in hardware except that its result is of an integer type instead of a floating-point type.

Some observers contended that `EXPONENT(0.0)` should not be 0; the most mathematically sensible alternative, $-\infty$, which can be represented on IEEE hardware at least, is ruled out by the integer-type result of `EXPONENT`. The committee staunchly preferred to stick with an integer type for the representation of the integer values delivered by this function, especially when it concluded that a result of zero for a zero argument is often a “don't care” case anyway (in the sense that the potential caller of `EXPONENT` will avoid the call and take a different path, when $X = 0.0$), and is probably harmless when not. Another alternative, raising an exception to signal an illegal argument when $X = 0.0$, was ruled out because it is unnecessarily harsh when a zero result is harmless.

The companion function `FRACTION` is also useful in argument reduction steps. For nonzero X , `FRACTION(X)` is defined to yield $X \cdot \beta^{-k}$, where k is as defined above for `EXPONENT`; `FRACTION(0.0)` is 0.0. Thus, `FRACTION(X)` is the fraction part of the canonical form of the floating-point number X (normalized, however, when X is denormalized). This function can be computed on typical hardware by extracting the fraction field of the representation, with a special case for $X = 0.0$ and with additional steps required when X is denormalized.

³For simplicity, this is understood to mean either actual underflow or merely denormalization, which is also known as “gradual underflow.”

⁴This is a reasonable assumption, without which some numbers expressible in the canonical form would not be representable. It requires, however, that the definition of canonical form be relaxed to allow unnormalized fraction parts.

Often, both the exponent part and the fraction part of a floating-point number are needed in argument reduction. For such occasions, the procedure `DECOMPOSE`, which computes and delivers both simultaneously through a pair of arguments of mode "out," is provided.

The function `COMPOSE` is essentially the inverse of `DECOMPOSE`; it constructs a floating-point value from a given fraction and exponent part. Except when `FRACTION = 0.0`, `COMPOSE(FRACTION, EXPONENT)`—for arguments of the floating-point type `FLOAT_TYPE` and the integer type `EXPONENT_TYPE`, respectively—delivers the value $\text{FRACTION} \cdot \beta^{\text{EXPONENT}-k}$ (if it is representable), where k is the unique integer such that $\beta^{k-1} \leq |\text{FRACTION}| < \beta^k$; `COMPOSE(0.0, EXPONENT)` delivers 0.0 for any `EXPONENT`. If the defined result is not representable, then the appropriate predefined exception is raised in overflow situations, and one of the adjacent representable numbers is delivered in underflow situations. Note that the `FRACTION` argument is not required to be a pure fraction, with a zero exponent part (as if it had been obtained from the `FRACTION` function previously); rather, the fraction part of `FRACTION` is extracted and used to construct the result. It should be obvious that this function can be computed, on typical hardware, by appropriate manipulations of the fraction and exponent parts of floating-point quantities, as for the previous functions. `COMPOSE` finds representative uses in the result construction step of mathematical functions.

The remaining function of the first group, `SCALE`, is similar to `COMPOSE`; it has uses both in result construction steps and in argument conditioning (for Euclidean norms, complex arithmetic, and some matrix computations, for example). It takes arguments `X` and `EXPONENT` and returns $X \cdot \beta^{\text{EXPONENT}}$ (with the same provisions for dealing with overflow and underflow as exhibited by `COMPOSE`). `SCALE` is analogous to the IEEE recommended function `scalb`. When implemented by directly manipulating the exponent part of a floating-point number, it is potentially more efficient than multiplying or dividing by a power of the hardware radix, and by definition it retains full accuracy (multiplication and division, even by a power of the hardware radix, can lose accuracy on systems lacking guard digits for these operations). The function is sometimes available as a hardware operation.

The functions of the first group are not all independent. In theory, it is sufficient to have just `EXPONENT` and `SCALE`, or alternatively `EXPONENT` and `COMPOSE`; the others can be obtained in terms of these two. For greater efficiency, however, implementations should code each independently using the most direct interface to low-level representations and operations available.

The second group of subprograms comprises directed rounding functions (`ROUND`, `TRUNCATE`, `FLOOR`, and `CEILING`, all of which yield an integer value in the floating-point type `FLOAT_TYPE`) and an exact remainder function (`REMAINDER`).

`ROUND`, of course, delivers the value of its argument, rounded to the nearest integer, with ties being broken by choosing the even integer; this corresponds to IEEE unbiased rounding. Ada already has something comparable in its predefined conversion between floating-point and integer types. The `ROUND` function differs in having a floating-point result type and in specifying that ties will be broken by choosing the even integer. `ROUND` and the other directed rounding functions are supposed to produce their floating-point results without going through an intermediate conversion to an integer type, which could raise an exception (often the higher-precision floating-point types can accommodate larger integer values than can be represented in the available integer type of widest range). `TRUNCATE` simply discards the fractional part, thereby rounding in the direction of zero. `FLOOR` and `CEILING` round in the negative and positive directions, respectively. All of these functions can be programmed efficiently at a low level and might even exist as hardware operations.

The `REMAINDER` function delivers the exact remainder upon dividing its first floating-point argument by its second floating-point argument. More precisely, `REMAINDER(X, Y)` finds an integer quotient q and a remainder r such that $r = X - Y \cdot q$; it delivers r . Algorithms exist for computing the result exactly, and reasonably efficiently, regardless of the relative magnitudes of the dividend and divisor; the operation is available as a hardware instruction on some machines.

There are two customary ways of defining the quotient q , which determines the corresponding remainder r . One way defines q as the integer obtained by rounding the exact value of X/Y towards zero. This gives r the sign of the dividend and a magnitude less than that of the divisor; it is the definition used by Ada for its predefined "rem" operator on integer-type operands, yielding an integer-type result. The other way defines q as the integer nearest the exact value of X/Y , with ties broken by choosing the even integer. This, in turn, gives r a magnitude not greater than half that of the divisor and a sign that may be either positive or negative. Because the latter definition, corresponding to the IEEE `rem` operation, is slightly preferred by numerical analysts for such purposes as argument reduction in the arbitrary-cycle versions of the trigonometric functions, it is what has been adopted for the primitive functions standard. It is tempting to offer this function in the form of an overloading of the predefined

"rem" operator, and indeed an earlier version of the proposed standard did so. However, the two overloaded "rem" operators would have distinctly different numerical behavior (e.g., `43 rem 5` yields 3, whereas `43.0 rem 5.0` would yield -2.0), so to avoid confusion the functional form `REMAINDER(X, Y)` was preferred for the floating-point remainder operation in `GENERIC_PRIMITIVE_FUNCTIONS`.⁵

The third group of subprograms contains the `PREDECESSOR`, `SUCCESSOR`, and `ADJACENT` functions, which allow a floating-point machine number to be perturbed by the smallest possible amount to obtain the next larger or smaller machine number. The principal use for these functions is in testing mathematical software, where very fine control over test arguments is sometimes needed. As defined, they are also useful for generating the machine numbers (denormalized, if the hardware has that capability) adjacent to zero.

`PREDECESSOR` and `SUCCESSOR` are one-argument functions that deliver the machine number adjacent to their argument in the direction inferred from the name, whereas `ADJACENT` is a two-argument function that returns the machine number adjacent to its first argument in the direction of the second argument. The latter function is provided for applications in which the direction of motion is not known in advance and needs to be determined dynamically; it is identical to the IEEE recommended function `nextafter`. There is another difference between `ADJACENT` and the other two functions: `PREDECESSOR` and `SUCCESSOR` raise the predefined exception signaling overflow upon an attempt to move beyond the first or last floating-point machine number, while `ADJACENT` never raises an exception (it is not possible to move beyond the range of machine numbers with it). The committee debated whether it was extravagant to have both sets and found itself split into two camps, neither of which wanted to give up its preferred choice. It was argued that one could not be assured of obtaining the other set if only one set were provided, because of a well-known weakness in the Ada model of floating-point arithmetic that makes the comparison of nearby floating-point numbers indeterminate.

The final group of subprograms contains miscellaneous functions—namely, `COPY_SIGN` and `LEADING_PART`.

The `COPY_SIGN` function, often found in other languages and represented in LCAS by `sign` and in the IEEE recommended functions by `copysign`, delivers the value obtained by transferring the sign of its second argument to the first (but otherwise retaining all the precision of the first argument). This function is often useful in giving the final result of some computation the appropriate sign (without resorting to an if-then-else test) after having stripped the sign away in the argument reduction step, perhaps by using the very same function to set it positive there. In highly accurate and portable code, this function is preferable to negation and the `abs` operator because those can lose low-order digits on hardware lacking a guard digit for subtraction. On hardware distinguishing the sign of zero (such as IEEE hardware), and where the implementation of `GENERIC_PRIMITIVE_FUNCTIONS` chooses to exploit the capability of signed zeros, `COPY_SIGN` is required to distinguish between plus zero and minus zero for its second argument; thus, it confers the sign of its second argument on the result even when the second argument is zero. `COPY_SIGN` was a late addition to `GENERIC_PRIMITIVE_FUNCTIONS`.

`LEADING_PART`, another late addition, was motivated by the LCAS `trunc` operation. It delivers the value of its first argument with only some of the leading radix-digits retained (the number of them given by the value of the second argument, which is of the predefined type `POSITIVE`), and with the remaining radix-digits—the low-order digits—replaced by zeros. This function plays a leading role in sophisticated strategies for simulating higher precision, where a floating-point number needs to be decomposed into a major portion of limited precision and an additive residue. The leading part is usually used as a factor in a subsequent multiplication by a small integer, such that the result has a sufficiently small number of radix-digits to be represented *exactly* within the model of floating-point arithmetic. The residue can be accurately obtained by subtraction, assuming the starting value has no more precision than that of safe numbers.

Two functions in the earliest versions of the proposed standard, both taken from [3], were dropped along the way. These were `RECIPROCAL_REL_SPACING(X)` and `ABS_SPACING(X)`, which give information about the spacing of machine numbers in the neighborhood of `X`. Although they are useful for fine control over the termination of an iterative algorithm, or for measuring and reporting error, committee members did not find them important enough to retain; when the committee was unable to justify their inclusion to the satisfaction of some observers, it decided to omit them.

The relationship between functions in `GENERIC_PRIMITIVE_FUNCTIONS` and certain required operations or recommended functions of the IEEE floating-point standards has been mentioned repeatedly in this rationale. It is

⁵The preceding discussion only scratches the surface of the long and involved history of this operation. Many other alternatives, some of which made their way into earlier drafts of the standard, were considered at one time or another.

anticipated that relevant functions in `GENERIC_PRIMITIVE_FUNCTIONS` will serve as the realization of some of the functionality of the proposed IEEE binding for Ada [4].

The relationship between functions in `GENERIC_PRIMITIVE_FUNCTIONS` and some of the features of LCAS has also been discussed. The fair degree of overlap between the two has prompted the suggestion that everything in LCAS that is not already built into Ada should be available in `GENERIC_PRIMITIVE_FUNCTIONS` in a compatibly defined way. The obvious benefit of following that suggestion to the letter would be the ability of an LCAS binding for Ada to point to `GENERIC_PRIMITIVE_FUNCTIONS` as the embodiment of that part of its functionality not built into Ada. Unfortunately, this goal was not expounded early enough in the development of `GENERIC_PRIMITIVE_FUNCTIONS`, and a few differences remain.

Several partial implementations of `GENERIC_PRIMITIVE_FUNCTIONS`, varying in the degree to which they exploit knowledge of the underlying machine, exist. Some of them have tried to be relatively general, that is, adaptable to different architectures by suitable choice of parameters; none have yet tried to be as efficient as possible.

Through a report [16] to the Ada 9X Requirements Team, the SIGAda Numerics Working Group has had an influence on Ada 9X as a result of the work it did in developing the proposed primitive and elementary functions standards. The report contains a discussion of the problems of writing high-quality, portable mathematical software; it included a number of Ada 9X revision requests aimed at solving some of these problems. One of the recommendations was to include the functionality of `GENERIC_PRIMITIVE_FUNCTIONS` in the Ada language, in the form of attributes (of the function kind). Several of the issues discussed in the report have been accepted by the Ada 9X Requirements Team as requirements for Ada 9X [2], including the incorporation of both elementary functions and primitive functions in optional annexes in Ada 9X.

References

- [1] ACM SIGAda Numerics Working Group. Proposed Standard for a Generic Package of Primitive Functions for Ada, December 1990. Draft 1.0.
- [2] Ada 9X Project. Ada 9X Requirements. Office of the Under Secretary of Defense for Acquisition, Washington, December 1990.
- [3] W. S. Brown and S. I. Feldman. Environment Parameters and Basic Functions for Floating-Point Computation. *TOMS*, 6(4):510-523, December 1980.
- [4] R. B. K. Dewar. Proposed Ada Interface for the IEEE Standard for Binary Floating-Point Arithmetic, August 1989. Version 4.
- [5] K. W. Dritz. Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada. ANL Report ANL-89/2 Rev. 1, Argonne National Laboratory, Argonne, Illinois, October 1989. A later (December 1990) revision is available from the author.
- [6] B. Ford. Parameterization of the Environment for Transportable Mathematical Software. *TOMS*, 4(2):100-103, June 1978.
- [7] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.
- [8] IEEE. IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std. 854-1987, IEEE, New York, 1987.
- [9] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, March 1989. Draft 1.0.
- [10] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, December 1990. Draft 1.2.
- [11] P. Naur. Machine Dependent Programming in Common Languages. *BIT*, 7:123-131, 1967.
- [12] M. Payne, C. Schaffert, and B. Wichmann. Proposal for a Language Compatible Arithmetic Standard. *SIGPLAN Notices*, 25(1):59-86, January 1990.
- [13] M. Payne, C. Schaffert, and B. Wichmann. Proposal for a Language Compatible Arithmetic Standard. *SIGNUM Newsletter*, 25(1):2-43, January 1990.
- [14] K. A. Redish and W. Ward. Environmental Enquiries for Numerical Analysis. *SIGNUM Newsletter*, 6(1):10-15, 1971.
- [15] J. Reid. Functions for Manipulating Floating-Point Numbers. *SIGNUM Newsletter*, 14(4):11-13, December 1979.
- [16] J. Squire. Special Study Report on Ada Numerical Issues. Unpublished material, October 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1990		3. REPORT TYPE AND DATES COVERED Final: June 1994	
4. TITLE AND SUBTITLE PROPOSED STANDARD FOR A GENERIC PACKAGE OF PRIMITIVE FUNCTIONS FOR ADA Draft 1.0 ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group				5. FUNDING NUMBERS PE: 0604711N PN: X1144-CC	
6. AUTHOR(S) G. Myers, et al.				8. PERFORMING ORGANIZATION REPORT NUMBER TD 2179	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Ocean Systems Center San Diego, California 92152-5000				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space and Naval Warfare Systems Command Code SPWR-06 Washington, DC 20363				11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The proposed standard for the Generic Package of Primitive Functions (GPPF) for Ada represents the work of a large number of people in both the United States and Europe who have collaborated to develop specifications for packages of Ada mathematical functions. GPPF is the specification for primitive functions and procedures for manipulating the fraction part and exponent part of machine numbers of the generic floating-point type. Additional functions are provided for directed rounding to a nearby integer, for computing an exact remainder, for determining the immediate neighbors of a floating-point machine number, for transferring the sign from one floating-point machine number to another, and for shortening a floating-point machine number to a specified number of leading radix digits.					
14. SUBJECT TERMS C ³ I Architecture computers software programming languages				15. NUMBER OF PAGES 33	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAME AS REPORT		

UNCLASSIFIED

21a. NAME OF RESPONSIBLE INDIVIDUAL G. Myers	21b. TELEPHONE (include Area Code) (619) 553-4136	21c. OFFICE SYMBOL Code 4104
---	--	---------------------------------

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 0274	Library	(2)
Code 0275	Archive/Stock	(6)
Code 413	G. B. Myers, Jr.	(25)

Defense Technical Information Center
Alexandria, VA 22304-6145 (4)

NCCOSC Washington Liaison Office
Washington, DC 20363-5100

Center for Naval Analyses
Alexandria, VA 22302-0268

Navy Acquisition, Research and Development
Information Center (NARDIC)
Arlington, VA 22244-5114

GIDEP Operations Center
Corona, CA 91718-8000